Design and Implementation of a Context-Sensitive, Flow-Sensitive Activity Analysis Algorithm for Automatic Differentiation

Jaewook Shin, Priyadarshini Malusare, and Paul D. Hovland

Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439
jaewook,malusare,hovland@mcs.anl.gov

Summary. Automatic differentiation (AD) has been expanding its role in scientific computing. While several AD tools have been actively developed and used, a wide range of problems remain to be solved. Activity analysis allows AD tools to generate derivative code for fewer variables, leading to a faster run time of the output code. This paper describes a new context-sensitive, flow-sensitive (CSFS) activity analysis, which is developed by extending an existing context-sensitive, flow-insensitive (CSFI) activity analysis. Our experiments with eight benchmarks show that the new CSFS activity analysis is more than 27 times slower but reduces 8 overestimations for the MIT General Circulation Model (MITgcm) and 1 for an ODE solver (c2) compared with the existing CSFI activity analysis implementation. Although the number of reduced overestimations looks small, the additionally identified passive variables may significantly reduce tedious human effort in maintaining a large code base such as MITgcm.

Key words: automatic differentiation, activity analysis

1 Introduction

Automatic differentiation (AD) is a promising technique in scientific computing because it provides many benefits such as accuracy of the differentiated code and the fast speed of differentiation. Interest in AD has led to the development of several AD tools, including some commercial software. AD tools take as input a mathematical function described in a programming language and generate as output a mathematical derivative of the input function. Sometimes, however, users are interested in a derivative of an input function for a subset of the output variables with respect to a subset of the input variables. Those input and output variables of interest are called *independent* and *dependent* variables, respectively, and are explicitly specified by users. When the independent and dependent variable sets are relatively small compared to the input and output variable sets of the function, the derivative code can run much faster by not executing the derivative code for the intermediate variables that are not contributing to the desired derivative values. Such variables are said to be *passive* (or inactive). The other variables whose derivatives must be computed are said to be *active*, and the analysis that identifies active variables is called *activity analysis*. Following [7], we say a variable is *varied* if it (transitively) depends on any independent variable and *useful* if any dependent

variable (transitively) depends on it; we say it is *active* if it is both varied and useful. Activity analysis is *flow-sensitive* if it takes into account the order of statements and the control flow structure of the given procedure and *context-sensitive* if it is an interprocedural analysis that considers only realizable call-return paths.

In our previous work, we developed a context-sensitive, flow-insensitive activity analysis algorithm, called variable dependence graph activity analysis (VDGAA), based on variable dependence graphs [11]. This algorithm is very fast and generates high-quality output; in other words, the set of active variables determined by the algorithm is close to the set of true active variables. However, we have observed a few cases where the algorithm overestimated passive variables as active because of its flow insensitivity. These cases suggest that the overestimations could be eliminated if we developed an algorithm that is both context-sensitive and flow-sensitive (CSFS). Such an algorithm would also be useful in evaluating overestimations of VDGAA.

Often, AD application source codes are maintained in two sets: the codes that need to be differentiated and those that are kept intact. When the codes in the former set are transformed by an AD tool, some passive global variables are often overestimated as active by the tool. If these global variables are also referenced by the codes in the latter set, type mismatch occurs between the declarations of the same global variable in two or more source files: the original passive type vs. AD transformed active type. Similar situations occur for functions when passive formal parameters are conservatively determined as active by AD tools and the functions are also called from the codes in the latter set [4]. In order to adjust the AD transformed types back to the original type, human intervention is necessary. As one option, users may choose to annotate such global variables and formal parameters in the code so that AD tools can preserve them as passive. However, this effort can be tedious if all formal variables that are mapped to the globals and formal parameters for the functions in the call chain have to be annotated manually. The burden of such human effort will be lifted significantly if a high-quality activity analysis algorithm is employed.

In this paper, we describe a CSFS activity analysis algorithm, which we have developed by extending VDGAA. To incorporate flow sensitivity, we use definitions and uses of variables obtained from UD-chains and DU-chains [1]. The graph we build for the new CSFS activity analysis is called def-use graph (DUG) because each node represents a definition now and each edge represents the use of the definition at the sink of the edge. Named after the graph, the new CSFS activity analysis algorithm is called DUGAA. The subsequent two sweeps over the graph are more or less identical to those in VDGAA. In a forward sweep representing the *varied* analysis, all nodes reachable from the independent variable nodes are colored red. In the following backward sweep representing the *useful* analysis, all red nodes reachable from any dependent variable node are colored blue. The variables of the blue nodes are also determined as *active*.

Our contributions in this paper are as follows:

- A new CSFS activity analysis algorithm
- Implementation and experimental evaluation of the new algorithm on eight benchmarks

In the next section, we describe the existing CSFI activity analysis VDGAA and use examples to motivate our research. In Section 3, the new CSFS activity analysis algorithm DUGAA is described. In Section 4, we present our implementation and experimental results. In Section 5, we discuss related research. We conclude and discuss future work in Section 6.

2 Background

We motivate our research by explaining the existing CSFI activity analysis algorithm and its flow insensitivity. We then discuss how flow sensitivity can be introduced to make a context-sensitive, flow-sensitive algorithm.

```
   subroutine head(x,y)
   double precision :: x,y
c$openad INDEPENDENT(x)
      call foo(x, y)
c$openad DEPENDENT(y)
   end subroutine
   subroutine foo(f,g)
      double precision :: f,g,a
      a = f
      g = a
   end subroutine
```

(a) All variables are active.

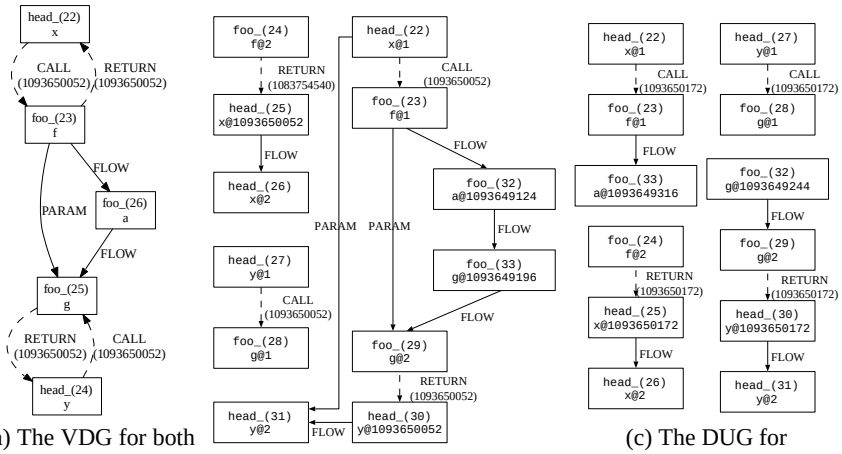
```
   subroutine head(x,y)
   double precision :: x,y
c$openad INDEPENDENT(x)
      call foo(x, y)
c$openad DEPENDENT(y)
   end subroutine
   subroutine foo(f,g)
      double precision :: f,g,a
      g = a
      a = f
   end subroutine
```

(b) No variables are active.

Fig. 1. Example showing the flow insensitivity of the existing CSFI algorithm.

VDGAA starts by building a variable dependence graph, where nodes represent variables and edges represent dependence between them [9]. Since a variable is represented by a single node in the graph, all definitions and uses of a variable are represented by the edges coming in and out the node. The order information among the definitions and uses cannot be retrieved from the graph. By building this graph, we assume that all definitions of a variable reach all uses of the variable. In terms of activity, this assumption results in more active variables than the true active ones. The two code examples in Figure 1 show the overestimation caused by flow insensitivity of VDGAA. In Figure 1(a), all five variables are active because there is a value flow path from x to y that includes all five variables, $x \rightarrow f \rightarrow a \rightarrow g \rightarrow y$, while no variables are active in (b) because no value flow paths exist from x to y.

Figure 2(a) shows a variable dependence graph generated by VDGAA, which produces the same graph for both codes in Figure 1. Nodes are connected with directed edges representing the direction of value flow. The edge labels show the edge types, which can be CALL, RETURN, FLOW, or PARAM. A pair of CALL and RETURN edges is generated for each pair of actual and formal parameters if called by reference. FLOW edges are generated for assignment statements, one for each pair of a used variable and a defined variable in the statement. PARAM edges summarize the value flows between formal parameters of procedures such that there is a PARAM edge from a formal parameter to another if there is a value flow path between them in the same direction. In Figure 2(a), two pairs of CALL and RETURN edges show the value flow between actual and formal parameters for the two actual parameters in the call to foo. The two FLOW edges are generated for the two assignment statements in procedure foo. The PARAM edge from node 23 to node 25 summarizes the value flow path $f \rightarrow a \rightarrow g$. Although not useful in this example, PARAM edges allow all other types of edges to be navigated only once during the subsequent *varied* and *useful* analyses. The numbers in the edge labels show the address of the call expression for CALL and RETURN edges, which are used to allow color propagations only through realizable control paths. Because of its flow insensitivity, the same graph is generated from the two different codes in Figure 1, and hence the same activity output. Although we know this behavior of VDGAA, determining the amount of overestimation is not easy.



(a) The VDG for both Figure 1(a) and (b).

(b) The DUG for Figure 1(a).

(c) The DUG for Figure 1(b).

Fig. 2. Def-use graphs generated by the new CSFS algorithm.

We developed a context-sensitive, flow-sensitive activity analysis algorithm to achieve two goals. First, we wish to evaluate how well VDGAA performs in terms of both the analysis run time and the number of active variables. Second, in the cases argued in Section 1, identifying several more inactive variables compared with VDGAA is desirable, even at the cost of the longer analysis time. The key idea in the new CSFS algorithm (DUGAA) is to use variable definitions obtained from UD/DU-chains [1] to represent the nodes in the graph. DUGAA combines flow sensitivity of UD/DU-chains with the context sensitivity of VDGAA. Since a statement may define more than one variable[1], as a node key we use a pair comprising a statement and a variable. Figures 2(b) and (c) show the two def-use graphs for the two codes in Figures 1(a) and (b). Unlike the VDG in Figure 2(a), the node labels in DUGs have a statement address concatenated at the end of the variable name and a symbol @. DUG is similar to *system dependence graph* of [10]. Among other differences, DUG does not have predicate nodes and control edges. Instead, flow sensitivity is supported by using UD/DU-chains. We use two special statement addresses: 1 and 2 for the incoming and outgoing formal parameter nodes, respectively. Since the DUG in Figure 2(c) has no path from any of the independent variable nodes (for x) to any of the dependent variable nodes (for y), no variables are active in the output produced by DUGAA for the code in Figure 1(b).

3 Algorithm

In this section, we describe the new activity analysis algorithm DUGAA. Similar to VDGAA, the DUGAA algorithm consists of three major steps:

1. Build a def-use graph.
2. Propagate red color forward from the independent variable nodes to find the *varied* nodes.
3. Propagate blue color backward along the red nodes from the dependent variable nodes to find the *active* nodes.

[1] as in call-by-reference procedure calls of Fortran 77

A def-use graph is a tuple (V, E), where a node N ∈ V represents a definition of a variable in a program and an edge (n1, n2) ∈ E represents a value flow from n1 to n2. Since all definitions of a variable are mapped to their own nodes, flow sensitivity is preserved in DUG.

```
Algorithm Build-DUG(program PROG)
  UDDUChain ← build UD/DU-chains from PROG
  DUG ← new Graph
  DepMatrix ← new Matrix
  for each procedure Proc ∈ CallGraph(PROG) in reverse postorder
    for each statement Stmt ∈ Proc
      // insert edges for the destination operand
      for each (Src,Dst) pair ∈ Stmt where Src and Dst are variables
        InsertUseDefEdge(Src, Dst, Stmt, Proc)
      // insert edges for the call sites in the statement
      for each call site Call to Callee ∈ Stmt
        for each (ActualVar,FormalVar) pair ∈ Call
          InsertCallRetEdges(ActualVar, FormalVar, Stmt, Proc, Callee, Call)
  connectGlobals()
  makeParamEdges()
```

Fig. 3. Algorithm: Build a def-use graph from the given program.

Figure 3 shows an algorithm to build a DUG. For each statement in a given program, we generate a FLOW edge from each reaching definition for each source variable to the definition of the statement. If the statement contains procedure calls, we also add CALL and RETURN edges. For global variables, we connect definitions after we process all statements in the program. PARAM edges are inserted between formal parameter nodes for each procedure if there is a value flow path between them. Below, each of the major component algorithms is described in detail.

```
Algorithm InsertUseDefEdge(variable Src, \
  variable Dst, stmt Stmt, procedure Proc)
  DefNode ← node(Stmt, Dst)
  // edges from uses to the def
  for each reaching definition Rd for Src
    // for an upward exposed use
    if (Rd is an upward exposed use)
      if (Src is a formal parameter)
        Rd ← stmt(1)
      else
        if (Src is a global variable)
          GlobalUpUse[Src].insert(aRecord( \
          Dst, Stmt, Proc, callExp(0), Proc))
        continue
    DUG.addEdge(node(Rd, Src), DefNode, \
    FLOW, Proc, Proc, Proc, callExp(0))
  // edges for downward exposed definitions
  if (Stmt has a downward exposed def)
    if (Dst is a formal parameter)
      DUG.addEdge(DefNode, node(stmt(2), \
      Dst), FLOW, Proc, Proc, Proc, callExp(0))
    else if (Dst is a global variable)
      GlobalDnDef[Dst].insert(aRecord(Dst, Stmt, \
      Proc, callExp(0), Proc))
```

```
Algorithm InsertCallRetEdge(variable Actual, \
  variable Formal, stmt Stmt, procedure Proc, \
  procedure Callee, callExp Call)
  // CALL edges from actuals to the formal
  for each reaching definition Rd for Actual
    if (Rd is an upward exposed use)
      if (Actual is a formal parameter)
        Rd ← stmt(1)
      else if (Actual is a global variable)
          GlobalUpUse[Actual].insert(aRecord(\
          Formal, stmt(1), Callee, Call, Proc))
        continue
    DUG.addEdge(node(Rd, Actual), node(stmt(1), \
    Formal), CALL, Proc, Callee, Proc, Call)
  // RETURN edges for call-by-reference parameters
  if (Actual is not passed by reference) return
  DUG.addEdge(node(stmt(2), Formal), node(Stmt, \
  Actual), RETURN, Callee, Proc, Proc, Call)
  // edges for downward exposed definitions of Actual
  if (Stmt has a downward exposed def) // DU-chain
    if (Actual is a formal parameter)
      DUG.addEdge(node(Stmt, Actual), node(stmt(2), \
      Actual), FLOW, Proc, Proc, Proc, callExp(0))
    else if (Actual is a global variable)
      GlobalDnDef[Actual].insert(aRecord(Actual, \
      Stmt, Proc, callExp(0), Proc))
```

Fig. 4. Algorithm: Insert edges.

Flow sensitivity is supported by using variable definitions obtained from UD/DU-chains. Since a statement may define multiple variables as in call-by-reference function calls, however, we use both statement address and variable symbol as a node key. We generate two nodes for each formal parameter: one for the incoming value along CALL edge and the other for the outgoing value along RETURN edge. As discussed in Section 2, two special statement addresses are used for the two formal parameter nodes. Upward exposed uses and downward exposed definitions must be connected properly to formal parameter nodes and global variable nodes. Figure 4 shows two algorithms to insert edges. $\mathrm{InsertUseDefEdge}$ inserts multiple edges for the given pair of a source variable (Src) and a destination variable (Dst) in an assignment statement (Stmt). UD-chains are used to find all reaching definitions for Src and to connect them to the definition of Dst. If the reaching definition is an upward exposed use, an edge is connected from an incoming node if Src is a formal parameter; if Src is a global variable, the corresponding definition (Dst and Stmt) is stored in $\mathrm{GlobalUpUse}$ for Src together with other information. If the definition of Dst is downward exposed, we connect an edge from the definition node to the outgoing formal parameter node if Dst is a formal parameter; for global Dst, we store the definition information in $\mathrm{GlobalDnDef}$. Later, we make connections from all downward exposed definitions to all upward exposed uses for each global variable. $\mathrm{InsertCallRetEdge}$ inserts edges between a pair of actual and formal parameter variables. CALL edges are inserted from each reaching definition of the actual parameter to the incoming node of the formal parameter. If the actual parameter is passed by reference, a RETURN edge is also inserted from the outgoing node of the formal parameter to the definition node of the actual parameter at Stmt.

```
Algorithm makeParamEdges()
  for each procedure Proc ∈ CallGraph(PROG) in postorder
    for each node N1 ∈ ProcNodes[Proc]
      for each node N2 ∈ ProcNodes[Proc]
        if (N1 == N2) continue
        if (DepMatrix[Proc][N1][N2]) continue
        if (!DUG.hasOutgoingPathThruGlobal(N1)) continue
        if (!DUG.hasIncomingPathThruGlobal(N2)) continue
        if (DUG.hasPath(N1, N2))
          DepMatrix[Proc][N1][N2] = true
    transitiveClosure(Proc)
    for each formal parameter Formal1 ∈ Proc
      for each formal parameter Formal2 ∈ Proc
        FNode1 ← node(stmt(1), Formal1)
        FNode2 ← node(stmt(2), Formal2)
        if (!DepMatrix[Proc][FNode1][FNode2]) continue
        DUG.addEdge(FNode1, FNode2, PARAM, Proc, Proc, Proc, callExp(0))
        for each call site Call ∈ Callsites[Proc]
          Caller ← CallsiteToProc[Call]
          for each node Actual2 ∈ FormalToActualSet[Call][FNode2]
            if (Actual2.Symbol is not called by reference) continue
            for each node Actual1 ∈ FormalToActualSet[Call][FNode1]
              DepMatrix[Caller][Actual1][Actual2] ← true
```

Fig. 5. Algorithm: Make PARAM edges.

PARAM edges summarize value flow among formal parameters to allow multiple traversals across formal parameter nodes when there are multiple call sites for the same procedure. We add a PARAM edge from an incoming formal parameter node $\mathrm{f1}$ to an outgoing formal parameter node $\mathrm{f2}$ whenever there is a value flow path from $\mathrm{f1}$ to $\mathrm{f2}$. Figure 5 shows the al-

gorithm that inserts PARAM edges. Whenever a FLOW edge is created, we set an element of the procedure's dependence matrix to true. After building a DUG for statements and connecting global variable nodes, for all pairs of formal parameters we check whether there is a value flow path between them going through other procedures via two global variables. This checking is necessary because we perform transitive closure only for those definitions used in each procedure. Next, we apply Floyd-Warshall's *transitive closure* algorithm [3] to find connectivity between all pairs of formal parameter nodes. A PARAM edge is added whenever there is a path from one formal node to another. We modified the original Floyd-Warshall's algorithm to exploit the sparsity of the matrix.

The *varied* and *useful* analyses are forward color propagation (with red) from the independent variable nodes and backward color propagation (with blue) from the dependent variable nodes, respectively. The propagation algorithms are described in our previous work [11].

4 Experiment

We implemented the algorithm described in Section 3 on OpenAnalysis [12] and linked it into an AD tool called OpenAD/F [13], which is a source-to-source translator for Fortran. Figure 6 shows the experimental flow. The generated AD tool was run on a machine with a 1.86 GHz Pentium M processor, 2 MB L2 cache, and 1 GB DRAM memory.

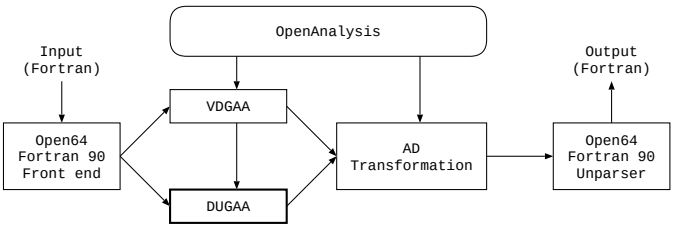


Fig. 6. OpenAD automatic differentiation tool.

Table 1. Benchmarks.

Benchmarks	Description	Source	# lines
MITgcm	MIT General Circulation Model	MIT	27376
LU	Lower-upper symmetric Gauss-Seidel	NASPB	5951
CG	Conjugate gradient	NASPB	2480
newton	Newton's method + Rosenbrock function	ANL	2189
adiabatic	Adiabatic flow model in chemical engineering	CMU	1009
msa	Minimal surface area problem	MINPACK-2	461
swirl	Swirling flow problem	MINPACK-2	355
c2	Ordinary differential equation solver	ANL	64

To evaluate our implementation, we used the set of eight benchmarks shown in Table 1. These benchmarks are identical to the ones used in our previous work [11] except for the version of the MIT General Circulation Model, which is about two times larger.

Figure 7 shows the slowdowns of DUGAA with respect to VDGAA, which are computed by dividing the DUGAA run times by the VDGAA run times. For *newton* and *c2*, the VDGAA run times were so small that the measurements were zero. For the other six benchmarks,

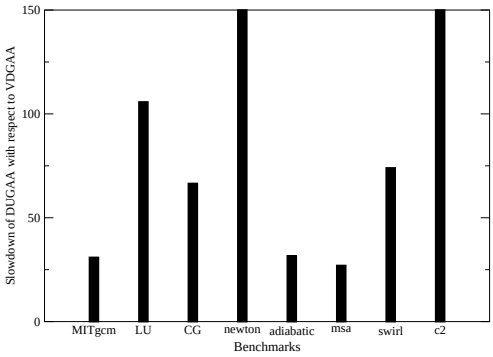


Fig. 7. Slowdown in analysis run time: DUGAA with respect to VDGAA.

the slowdowns range between 27 and 106. The benchmarks are ordered in decreasing order of program sizes, but the correlation with the slowdowns is not apparent. The run time for DUGAA on MITgcm is 52.82 seconds, while it is 1.71 seconds for VDGAA. Figure 8 show the component run times for both DUGAA and VDGAA on MITgcm. Since VDGAA does not use UD/DU-chains, the run time for computing UD/DU-chains is zero. However, it take 81.26% of the total run time for DUGAA. Another component worthy of note is *transitive closure*, which summarizes connectivity by adding PARAM edges between formal parameters. The transitive closure time can be considered as part of graph building but we separated it from the graph building time because it is expected to take a large portion. With respect to transitive closure times, the slowdown factor was 9.39. The graph navigation time for coloring was very small for both algorithms. The slower speed of DUGAA was expected because it would have many more nodes than VDGAA; The DUG for MITgcm has 13,753 nodes, whereas the VDG has 5,643 nodes.

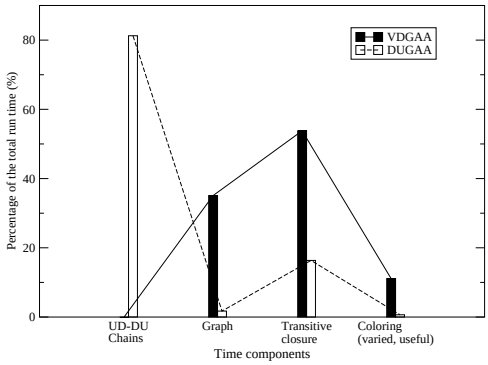


Fig. 8. Analysis run-time breakdown on MITgcm: DUGAA vs. VDGAA.

Our next interest is the accuracy of the produced outputs. Except for MITgcm and c2, the active variables determined by the two algorithms match exactly. Even for MITgcm and c2, the number of overestimations by VDGAA over DUGAA is not significant; 8 out of 925 for MITgcm and 1 out of 6 for c2. This result suggests several possibilities: First, as expected, the number of overestimations from flow insensitivity is not significant. Second, the flow sensitivity of DUGAA can be improved by having more precise UD/DU-chains. For example, actual parameters passed by reference are conservatively assumed to be nonscalar type. Hence, the definition of the corresponding scalar formal parameters does not kill the definitions coming from above. Third, aside from flow sensitivity, other types of overestimations can be made in both algorithms because they share important features such as graph navigation. One type of overestimation filtered by DUGAA is activating formal parameters when they have no edges leading to other active variables except to the corresponding actual parameters. Currently, VDGAA filters out the cases when the formal parameters do not have any outgoing edges than the RETURN edge going back to the actual parameter where the color is propagated from, but it fails to do so when there are other outgoing edges to other passive variables. This type of overestimation is filtered effectively by DUGAA by separating formal parameter nodes into two: an incoming node and an outgoing node. Although the number of reduced overestimations looks small, as argued in Section 1 the additionally identified passive variables may significantly reduce tedious human effort in maintaining a large code base such as MITgcm.

5 Related Work

Activity analysis is described in literature [2, 6] and implemented in many AD tools [5, 8, 11]. Hascoet et al. have developed a flow-sensitive algorithm based on iterative dataflow analysis framework [7]. Fagan and Carle compared the static and dynamic activity analyses in ADIFOR 3.0 [5]. Their static activity analysis is context-sensitive but flow-insensitive. Unlike other work, this paper describes a new context-sensitive, flow-sensitive activity analysis algorithm. Our approach of forward and backward coloring is similar to program slicing and chopping [14, 10]. However, the goal in that paper is to identify all program elements that might affect a variable at a program point.

6 Conclusion

Fast run time and high accuracy in the output are two important qualities for activity analysis algorithms. In this paper, we described a new context-sensitive, flow-sensitive activity analysis algorithm, called DUGAA. In comparison with our previous context-sensitive, flow-insensitive (CSFI) algorithm on eight benchmarks, DUGAA is more than 27 times slower but reduces 8 out of 925 and 1 out of 6, determined active by the CSFI algorithm for the MIT General Circulation Model and an ODE solver, respectively. We argue that this seemingly small reduction in number of active variables may significantly reduce tedious human effort in maintaining a large code base.

The current implementations for both DUGAA and VDGAA can be improved in several ways. First, if the nodes for the variables with integral types are not included in the graph, we expect that both the run time and the output quality can be improved. Second, more precise UD/DU-chains also can improve the output accuracy. Third, we might be able to identify other types of overestimation different from those already identified. Fourth, both VDGAA and DUGAA currently support only Fortran 77. Supporting Fortran 90 and C is left as a future work.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. We thank Gail Pieper for proofreading several revisions.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.
3. Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. McGraw Hill, 2nd edition, 1990.
4. Michael Fagan, Laurent Hascoet, and Jean Utke. Data representation alternatives in semantically augmented numerical models. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 85–94, Los Alamitos, CA, 2006. IEEE Computer Society.
5. Mike Fagan and Alan Carle. Activity analysis in ADIFOR: Algorithms and effectiveness. Technical Report TR04-21, Department of Computational and Applied Mathematics, Rice University, Houston, TX, November 2004.
6. Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.
7. Laurent Hascoët, Uwe Naumann, and Valérie Pascual. "To be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8):1401–1417, 2005.
8. Barbara Kreaseck, Luis Ramos, Scott Easterday, Michelle Strout, and Paul Hovland. Hybrid static/dynamic activity analysis. In *Proceedings of the 3rd International Workshop on Automatic Differentiation Tools and Applications (ADTA'04)*, Reading, England, 2006.
9. Arun Lakhotia. Rule-based approach to computing module cohesion. In *Proceedings of the 15th International Conference on Software Engineering*, pages 35–44, Baltimore, MD, 1993.
10. Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 41–52, 1995.
11. Jaewook Shin and Paul D. Hovland. Comparison of two activity analyses for automatic differentiation: Context-sensitive flow-insensitive vs. context-insensitive flow-sensitive. In *ACM Symposium on Applied Computing*, pages 1323–1329, Seoul, Korea, March 2007.
12. Michelle Mills Strout, John Mellor-Crummey, and Paul D. Hovland. Representation-independent program analysis. In *Proceedings of The Sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, September 2005.
13. Jean Utke. OpenAD: Algorithm implementation user guide. Technical Memorandum ANL/MCS–TM–274, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 2004. ftp://info.mcs.anl.gov/pub/tech_reports/reports/TM-274.pdf.
14. Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.